CITS2200 Data Structures and Algorithms

Topic 18

Search

- Dictionary Specifications
- Dictionary specifications
- Dictionary representations — set-based representations, search trees
- Self-balancing Search Trees

Reading: Weiss, Chapter 19.

© Tim French

1. Dictionary Specification

☐ **Constructors**

1. *Dictionary()*: creates an empty dictionary.

☐ **Checkers**

2. *isEmpty()*: returns *true* if the dictionary is empty, *false* otherwise.
3. *isMember(e)*: returns *true* if *e* is a member of the dictionary, *false* otherwise.
4. *isPredecessor(e)*: returns *true* if there is an element in the dictionary that precedes *e* in the total order, *false* otherwise.
5. *isSuccessor(e)*: returns *true* if there is an element in the dictionary that succeeds *e* in the total order, *false* otherwise.

© Tim French

☐ **Manipulators**

6. *insert(e)*: adds e (if not already present) to the dictionary in the appropriate position.
7. *predecessor(e)*: returns the largest element in the dictionary that is smaller than *e* if one exists, otherwise throws an exception.
8. *successor(e)*: returns the smallest element in the dictionary that is larger than *e* if one exists, otherwise throws an exception.
9. *range(p,s)*: returns the dictionary of all elements that lie between *p* and *s* (including *p* and *s* if present) in the ordering.
10. *delete(e)*: removes item e from the dictionary if it exists.

© Tim French

2. Dictionary Representations

2.1 Representations Based on Set

We have already seen two representations that can be used for Sets when there is a total ordering on the universe. . .

- characteristic function (bit vector) representation
  - time efficiency (eg $O(1)$ for *isMember*) gained by indexing directly to appropriate bits
  - bounded — universe fixed in advance
  - space wasted if universe is large compared with commonly occurring sets

© Tim French

- List based (ordered block) representation
  - time efficiency (eg $O(\log n)$ for *isMember*) comes from binary search
  - bounded
  - space usage may be poor if large block is set aside

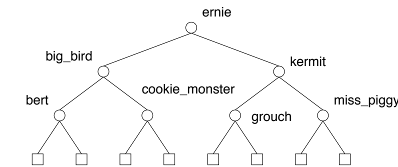
We now examine a representation which supports a binary-like search but is unbounded...

2.2 Binary Search Trees

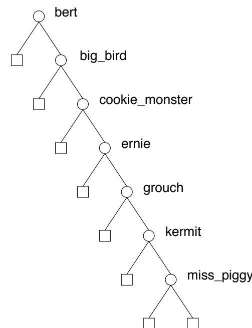
A *binary search tree* is a binary tree whose internal nodes are labelled with elements (or their keys) such that they satisfy the *binary search tree condition*:

> For every internal node u, all nodes in u's left subtree precede u in the ordering and all nodes in u's right subtree succeed u in the ordering.

eg.



eg.



Searching

If information is stored in a binary search tree, a simple recursive "divide and conquer" algorithm can be used to find elements:

```
if (t.isEmpty()) terminate unsuccessfully;
else {
  r becomes the element on the root node of t;
  if (e equals r) terminate successfully;
  else if (e < r) repeat search on left subtree;
  else repeat search on right subtree;
}
```

Performance

Depends on the shape of the tree...

Exercise

- Best case is a perfect binary tree. What is the performance of *isMember*?

- Worst case is a skinny binary tree. What is the performance of *isMember*?

insert and delete

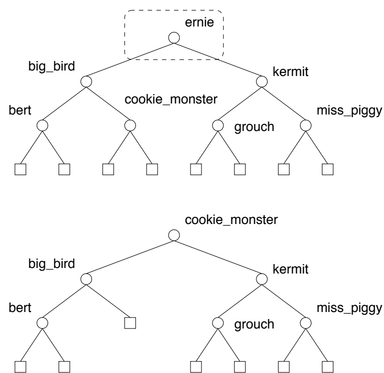
insert is fairly straightforward

- perform a search for the element as above
- if the element is found, take no further action
- if an empty node is reached, insert a new node containing the element

delete is straightforward if the element is found on a node with at least one external child — just use the standard Bintree *delete* operation

Otherwise:

1. replace the deleted element with its predecessor — note that the predecessor will always have an empty right child
2. delete the predecessor



Balancing Trees

Note that the *delete* procedure described here has a tendency over time to skew the tree to the right — as we have seen this will make it less efficient.

Alternative: alternate between replacing with predecessor and successor.

In general, it is beneficial to try to keep the tree as "balanced" or "complete" as possible to maintain search efficiency.

There are a number of data structures that are designed to keep trees balanced — *B-trees*, *AVL-trees* and *Red-black trees*.

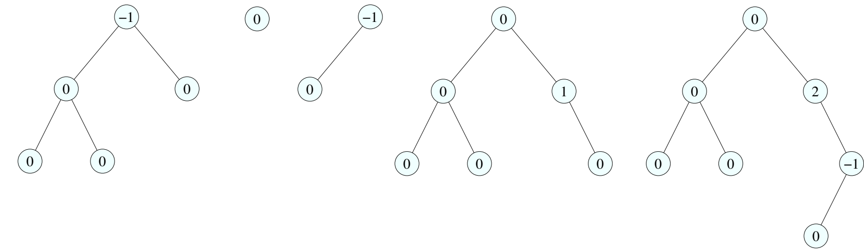
Note: Java.util.TreeMap uses Red-Black trees in its implementation. In general, for search trees with guaranteed logarithmic performance, it is better to use this class than to write your own.

2.3 AVL Trees

We will look at the three types of *self-balancing* trees, but we will only examine the operations of AVL trees. (The other trees just involve more complicated versions of these operations).

An *AVL tree* is a binary search tree where, for every node, the height of the left and right subtrees differ by at most one. This means the depth of any external node is no more than twice the depth of any other internal node.

The picture below demonstrates some AVL-trees. Nodes are marked with the height of the right sub-tree minus the height of the left sub-tree.



AVL trees have $O(\log n)$ time peformance for searching, inserting, and deleting.

AVL Tree Operations

Since an AVL tree *is* a binary search tree, the searching algorithm is exactly the same as for a binary tree.

However, the insertion and deletion operations must be modified to maintain the balance of the tree.

AVL Tree Insertions

We first find the appropriate place (a leaf node) to add the new element. If the insertion makes the tree unbalanced, then we locally reorganize the tree (via a *rotation*) to restore balance.

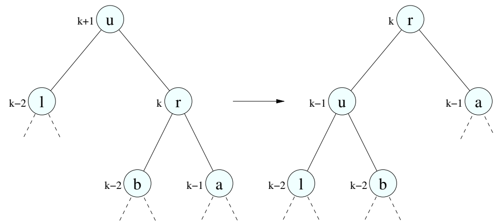
To insert an element, we:

1. Insert the element into an external node (as per usual for a binary search tree).
2. Starting with its parent, check each ancestor of the new node to ensure the left and right subtree heights differ by less than two.
3. If we find a node such that one child has height $k-2$ and the other has height k, then we perform a rotation to restore balance.

Rotation Case I

Suppose u is a node where its left child l has height two less than its right child r, and the *right* child of r has height one more than the *left* child of r.

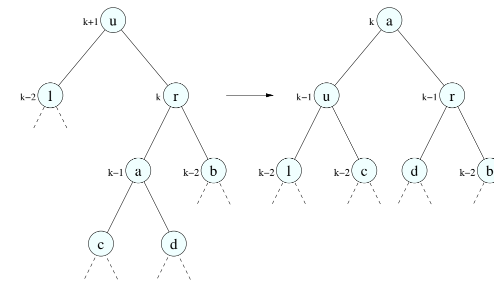
We rearrange the tree as follows:



Rotation Case II

Suppose u is a node where its left child l has height two less than its right child r, and the *left* child of r has height one more than the *right* child of r.

We rearrange the tree as follows:



AVL Tree Deletions

Note that both rotations do not increase the height of the sub-tree, so insertion only needs to be done at the lowest unbalanced ancestor.

To delete an element, we:

- Delete the element (as per usual for a binary search tree).
- Starting with its parent, check each ancestor of the new node to make sure it's balanced.
- If any node is not balanced, perform the necessary rotation (as above).
- Continue to check the ancestors of the deleted node up to the root.

Complexity

Rotations are constant time operations.

Insertions and deletions involve searching the tree for the element ($O(h)$, where h is the height of the tree) and then checking every ancestor of that element ($O(h)$ in the worst case).

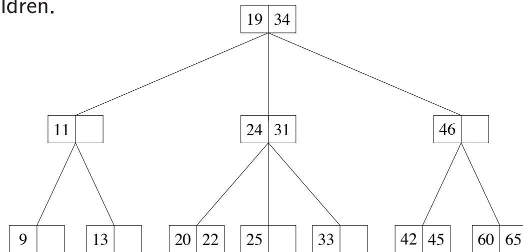
Complexity follows from the claim: *the height of an AVL tree is less than* $2 \log n$ *where* n *is the number of elements in the tree.*

2.4 B-trees

A *B-tree* is a tree data structure that allows searching, insertion, and deletion in amortized logarithmic time.

Each node of a B-tree can contain multiple items and multiple children (these numbers can be varied to tweak performance).

We will consider 2-3 B-trees, where each node can contain up to two items and up to three children.



If there is just one item in the node, then the B-Tree is organised as a binary search tree: all items in the left sub-tree must be less than the item in the node, and all items in the right sub-tree must be greater.

It there are two elements in the node, then:

- all items in the left sub-tree must be less than the smallest item in the node
- all items in the middle sub-tree must be between the two items in the node
- all elements in the right sub-tree must be greater than the largest item in the node

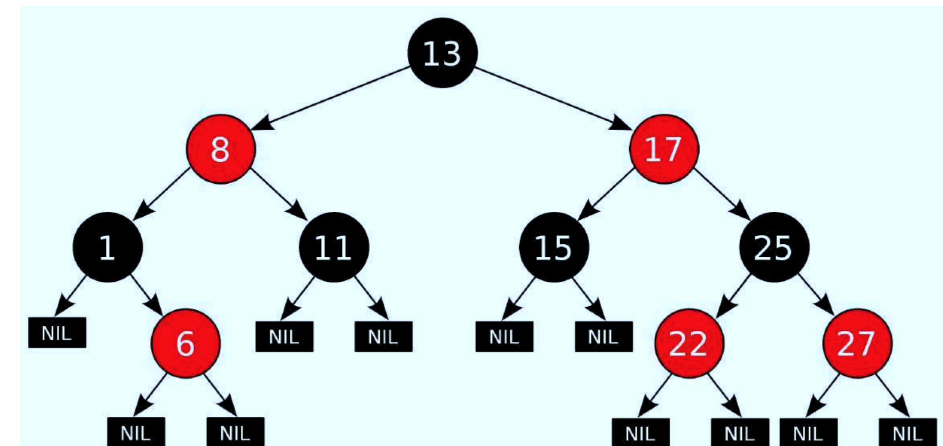
Also, every non-leaf node must have at least two successors and all leaf nodes must be at the same level.

2.5 Red-black Trees

A *red-black tree* is another variation of a binary search tree that is slightly more efficient (and complicated) than B-trees and AVL trees.

A red-black tree is a binary tree where each node is coloured either red or black such that the colouring satisfies:

- *the root property* — the root is black
- *the external property* — every external node is black
- *the internal property* — the children of a red node are black
- *the depth property* — every external node has the same number of black ancestors.



Picture courtesy of Wikimedia Commons

3. Summary

We have outlined 3 ADTs for use with collections of unique elements or records:

- Set — includes set-theoretic operations, elements may or may not be ordered
- Table — restriction of Set with fewer operations, elements assumed not ordered
- Dictionary — extension of Table, assumes ordering and contains "order-related" operations

We have covered a number of representations:

- List — can be used for unordered sets and tables
- ordered list (block) — can improve efficiency for ordered sets and can be used for bounded dictionaries
- characteristic function — can be very efficient for ordered sets and dictionaries in certain cases, bounded
- binary search tree — unbounded representation for dictionaries, efficiency better than List but depends on tree shape

Next — efficient representations for Tables...